Carnegie Mellon University
Software Engineering Institute

Serpent

C Application Developer's Guide

System for User Interface Development

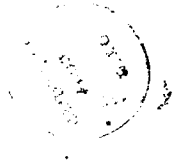
Version 1

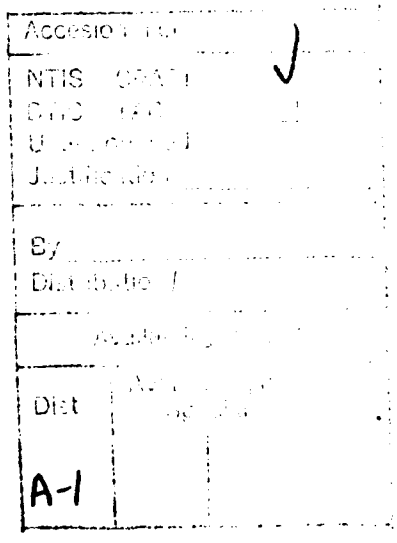
Date April 1991

User's Guide

May 1991
CMU/SEI-91-UG-6

Serpent: C Application Developer's Guide





User Interface Project

Approved for public release.
Distribution unlimited.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This document was prepared for the

SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

The ideas and findings in this document should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This document has been reviewed and is approved for publication.

FOR THE COMMANDER

Charles J. Ryan, Major, USAF
SEI Joint Program Office

The Software Engineering Institute is sponsored by the U.S. Department of Defense.
This report was funded by the Department of Defense.

Table of Contents

List of Figures

List of Examples

1 Introduction

Serpent is a user interface management system (UIMS) that supports the development and execution of a user interface of a software system. Serpent supports incremental development of the user interface from the prototyping phase through production to maintenance or sustaining engineering. Serpent encourages a separation of functionality between the user interface functional portions of a software system. Serpent is also easily extended to support additional user interface toolkits.

1.1 This Manual

This manual describes how to develop applications using Serpent. Readers are assumed to have read and understood the concepts described in the *Serpent Overview*, as well as to have had experience using the C programming language.

1.1.1 Organization

The contents of this guide include:

- **Introduction and Overview.** This chapter provides a general description of the role of an application in a software system developed with Serpent. It also describes a conceptual framework for application development.
- **Specifying the Contract.** This chapter describes the tasks necessary to define the type, structure and values of data to be shared between an application program and Serpent and to establish runtime communications with Serpent.
- **Modifying Information.** This chapter describes the tasks necessary to add, modify or remove information to/from the Serpent shared database.
- **Retrieving Information.** This chapter describes the tasks necessary to define and retrieve changes to information from the Serpent shared database.
- **Finishing the Application.** This chapter describes the finishing touches that should be applied to the application, including error checking and exception handling.
- **Testing and Debugging.** This chapter describes utilities available to assist in the testing and debugging of the application.
- **Appendix A: Data Structures.** This appendix is a complete reference of all the constants, types, routines, and other data structures available to Serpent application developers using the C programming language.

- **Appendix B: Routines.** This appendix is a complete reference of all the routines available to Serpent application developers using the C programming language.
- **Appendix C: Commands for Testing Serpent Applications and Dialogues.** This appendix is a reference of commands available to Serpent application developers from the operating system.
- **Appendix D: Spider Example.** This appendix is a complete application example, developed in the C programming language.

1.1.2 Typographical Conventions

Code examples	`Courier typeface`
Code directly related to text	**`Bold, courier typeface`**
Variables, attributes, etc.	`Courier typeface`
Syntax	`Courier typeface`
Warnings and cautions	***Bold, italics***

1.2 Other Serpent Documents

The purpose of this guide is to provide the information necessary to develop Serpent applications. The following publications address other aspects of Serpent.

Serpent Overview

Introduces the Serpent system.

Serpent: System Guide

Describes installation procedures, specific input/output file descriptions for intermediate sites, and other information necessary to set up a Serpent application.

Serpent: Saddle User's Guide

Describes the language that is used to specify interfaces between an application and Serpent.

Serpent: Dialogue Editor User's Guide

Describes how to use the editor to develop and maintain a dialogue.

Serpent: Slang Reference Manual

Provides a complete reference to Slang, the language used to specify a dialogue.

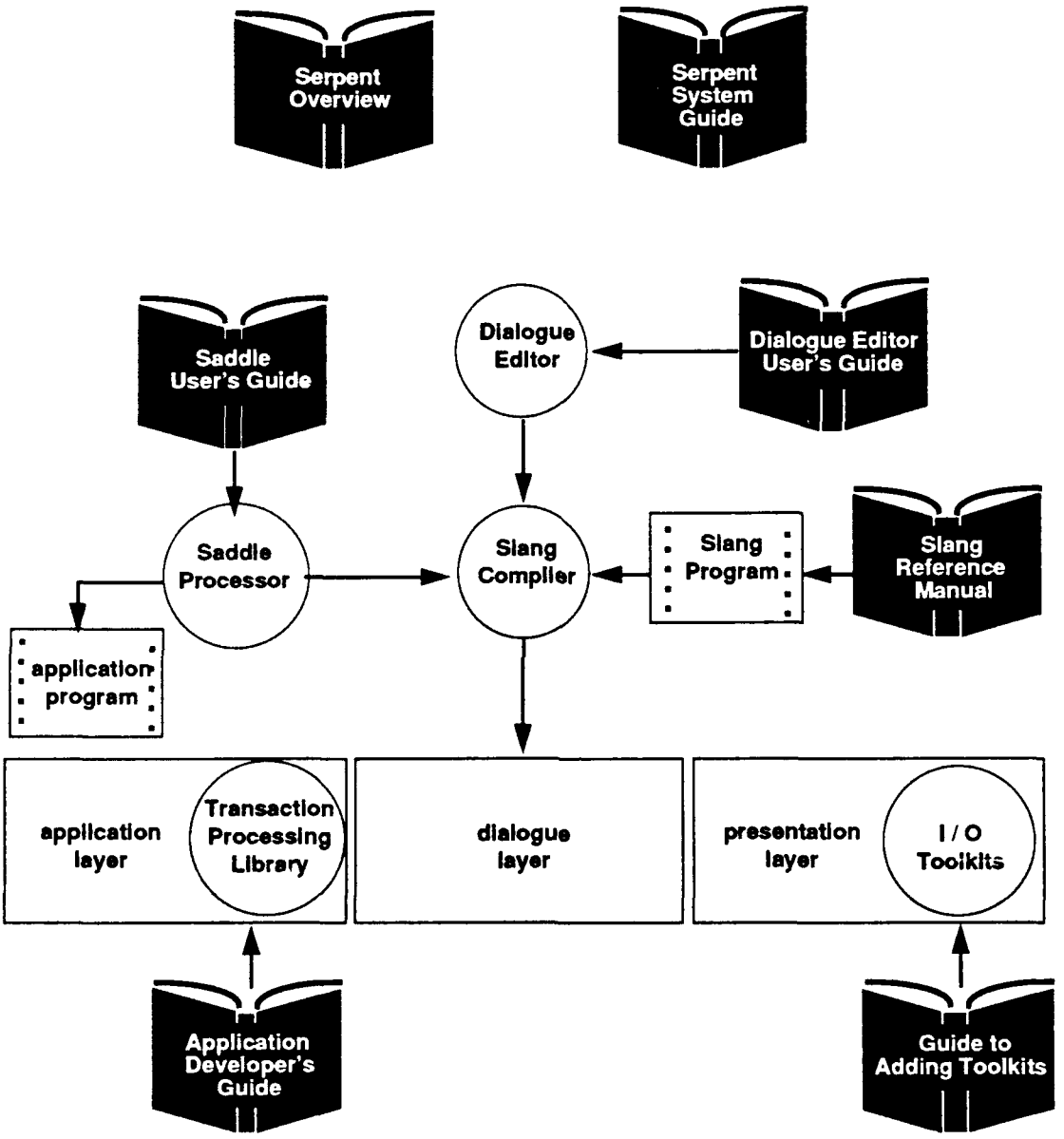
Serpent: Ada Application Developer's Guide

Describes how the application interacts with Serpent. This guide describes the runtime interface library, which includes routines that manage such functions as timing, notification of actions, and identification of specific instances of the data.

Serpent: Guide to Adding Toolkits

Describes how to add user interface toolkits, such as various Xt-based widget sets, to Serpent or to an existing Serpent application. Currently, Serpent includes bindings to the Athena Widget Set and the Motif Widget Set.

The following figure shows Serpent documentation in relation to the Serpent system:



2 Overview

A main goal of Serpent is to encourage the separation of a software system into an application portion and a user interface portion to provide the application developer with a presentation-independent interface. The application portion consists of those components of a software system that implement the "core" application functionality of a system. The user interface portion consists of those components that implement an end-user dialogue. A dialogue is a specification of the presentation of application information and end-user interactions.

During the design stage, the system designer decides which functions belong in the application component and which belong in the user interface component of the system.

2.1 Serpent Architecture

Serpent is implemented using a standard UIMS architecture. This architecture (see Figure 2-1) consists of three major layers: the *presentation layer*, the *dialogue layer*, and the *application layer*. The three different layers of the standard architecture are viewed as providing differing levels of end-user feedback.

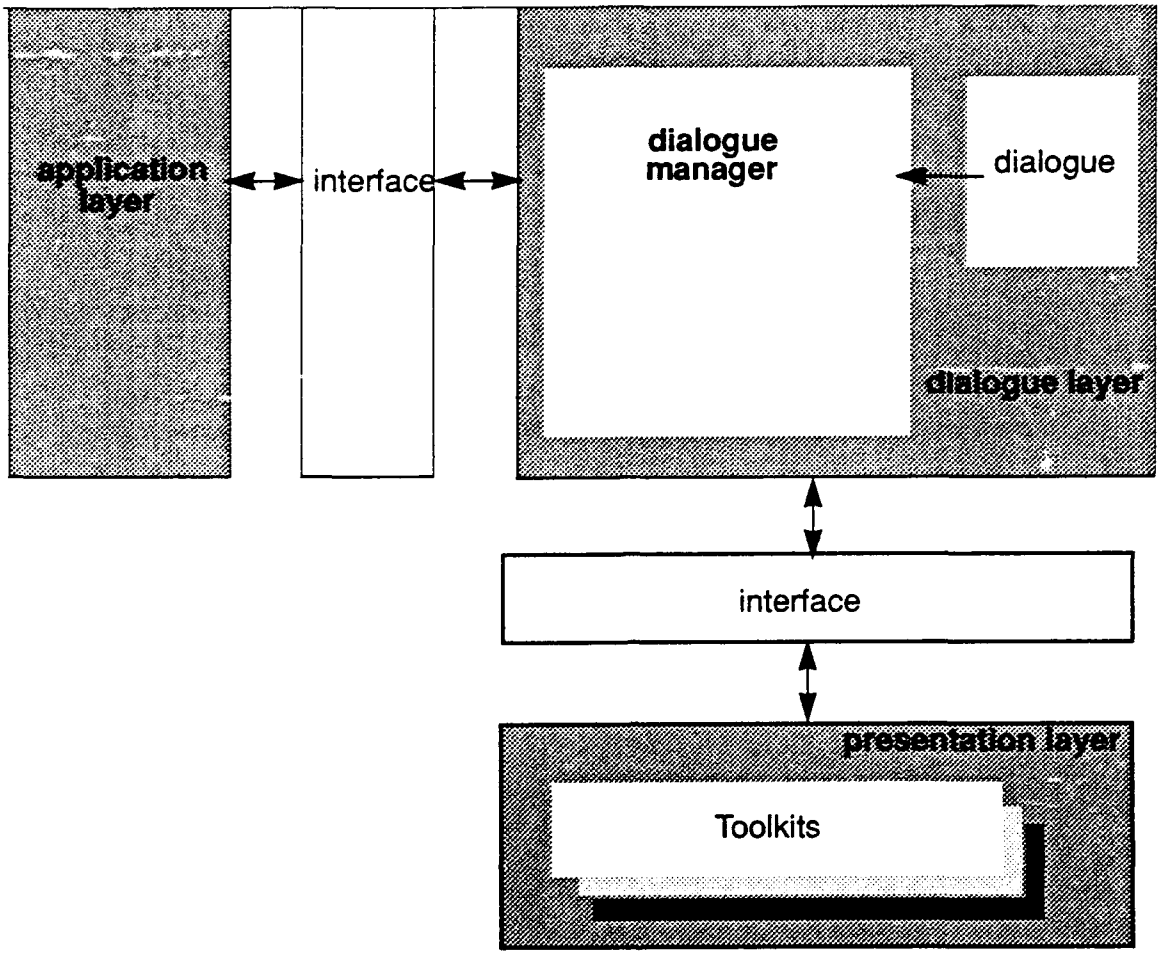


Figure 2-1 Serpent Architecture

The presentation layer consists of various input/output toolkits that have been incorporated into Serpent. Input/output toolkits are existing hardware/software systems that perform some level of generalized interaction with the end user. Serpent is being distributed with an interface to the X Window System, Version 11. Other input/output toolkits can be integrated with Serpent. See *Serpent: Guide to Adding Toolkits* for a discussion of how this can be accomplished.

One way of viewing the three levels of the architecture is the level of functionality provided for user input. The presentation layer is responsible for lexical functionality, the dialogue layer for syntactic functionality, and the application layer for semantic functionality. In terms of a menu example, the presentation layer has responsibility for determining which menu item was selected and for presenting feedback that indicates which choice is currently selected. The dialogue layer has responsibility for deciding whether another menu is presented and presenting it, or whether the choice requires application action. The application layer is responsible for implementing the command implied by the menu selection.

The end user interface for a software system is specified formally as a *dialogue*. The dialogue is executed by the dialogue manager at runtime in order to provide an end user interface for a software system. The dialogue specifies both the presentation of application information and end user interactions. The Serpent dialogue specification language (Slang) allows dialogues to be arbitrarily complex.

The application provides the functional portion of the software system in a presentation-independent manner. It may be developed in C, Ada, or other programming languages. The application may be either a functional simulation for prototyping purposes or the actual application in a delivered system. The actions of the application layer are based upon knowledge of the specific problem domain.

2.2 Shared Database

Serpent provides an active database model for specifying the user interface portion of a system. In an active database, multiple processes are allowed to update a database. Changes to the database are then propagated to each user of the database. This active database model is implemented in Serpent by a *shared database* that logically exists between the application and I/O toolkits. The application can add, modify, query, or remove data from the shared database. Information provided to Serpent by the application is available for presentation to the end user. The application has no knowledge of the presentation media or user interface styles used to present this information.

Information in the shared database may be updated by either the application or I/O toolkits. Figure 2-2 illustrates the use of the shared database in Serpent.

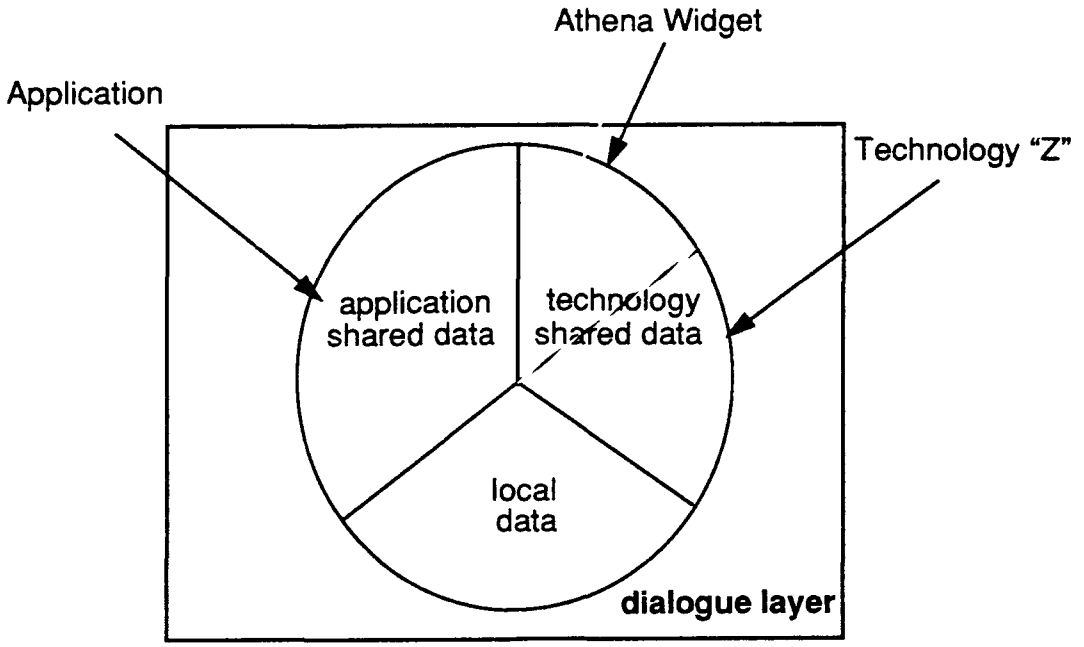


Figure 2-2 Shared Database

Serpent allows the specification of dependencies between elements in the shared database in the dialogue. These dependencies define a mapping among application data, presentation objects, and end user input. The dialogue manager enforces these dependencies by operating on the information stored in the shared database until the dependencies are met. Changes are then propagated to either the application or the I/O toolkits as appropriate. See the *Serpent: Slang Reference Manual* (CMU/SEI-91-UG-5) for a further discussion.

The *type* and *structure* of information that can be maintained in the shared database is defined externally in a *shared data definition file*. This corresponds to the database concept of *schemas*. A shared data definition file is required for each application.

A shared data definition file consists of both aggregate and scalar data structures. Top-level data structures become *shared data* elements that may be instantiated at runtime. Nested data structures become components that are considered part of the shared data element. Serpent does not allow nesting of records.

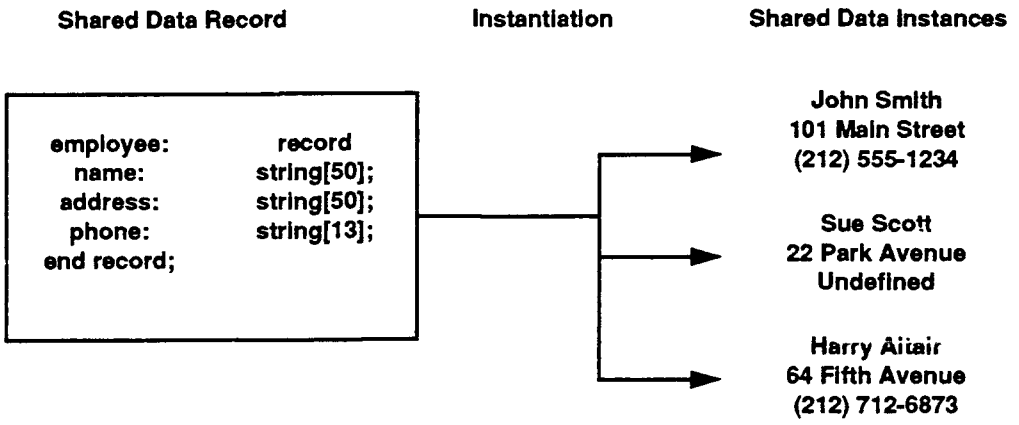


Figure 2-3 Shared Data Instantiation

It is possible to define multiple instances of a single shared data element. Shared data elements are instantiated by specifying the element name. Each *shared data instance* is identified by a unique *ID*. IDs must be maintained by the application to identify shared data instances when multiple instances of a single shared data element exist. Figure 2-3 provides an illustration of shared data instantiation.

Since the dialogue manager, the application, and any toolkits participating in a particular execution of Serpent are separate system processes that use the shared database, they can potentially modify the database concurrently, possibly compromising the integrity of the database. This problem is solved in Serpent through the use of database concurrency control techniques. Updates to the Serpent shared database are packaged in transactions. Transactions are collections of updates to the shared database that are logically processed at one time. Transactions can be *started*, *committed*, or *aborted*. A transaction which has been started but neither committed nor aborted yet is said to be *open*. Multiple transactions may be open at the same time. Committing a transaction causes the updates to be made to the shared database. Aborting a transaction causes termination of the transaction without any update of the shared database.

Communicating with Serpent

The application communicates with Serpent using the shared database model described earlier in this document. Information added to shared data is available to be presented to the end user by the dialogue. Changes to application data are automatically communicated back to the application.

2.3 Application Development

The application, or non-user interface portion of the system, provides the "core" functionality of a software system developed using Serpent. The application can be written in Ada, C, or other programming languages and can be either a simulation or an actual application.

An application may only add information to shared data or it may only retrieve information from shared data. For example, an application that monitors and displays the status of a computer network may only need to add information to shared data to update the display. An application such as an automatic teller machine (ATM) might only need to retrieve data from the user interface.

All transactions to and from the application are handled explicitly in the application using the routines and data structures available in the Serpent application interface. This document describes the usage and definitions of these routines and data structures.

Error Checking and Recovery

Each routine in Serpent sets status on exiting. It is the responsibility of the application developer to check this status to perform appropriate error recovery. Serpent provides routines to both check and print the status.

Testing and Debugging

Serpent provides a record/playback feature that can be used in testing and debugging. Transactions between the application and dialogue manager or between the dialogue manager and the various toolkits can be recorded, then played back at a later time. This is useful in isolating problems or in performing regression/stress testing of an application, dialogue, or toolkit.

Spider Example

The spider application is an example of an application system developed using Serpent. Figure 2-4 is an illustration of a "spider chart" display that is one possible end-user interface for the application.

Adapted from a command and control application, the spider application monitors and displays the status of various sensor sites and their associated communication lines to the two correlation centers (Figure 2-4).

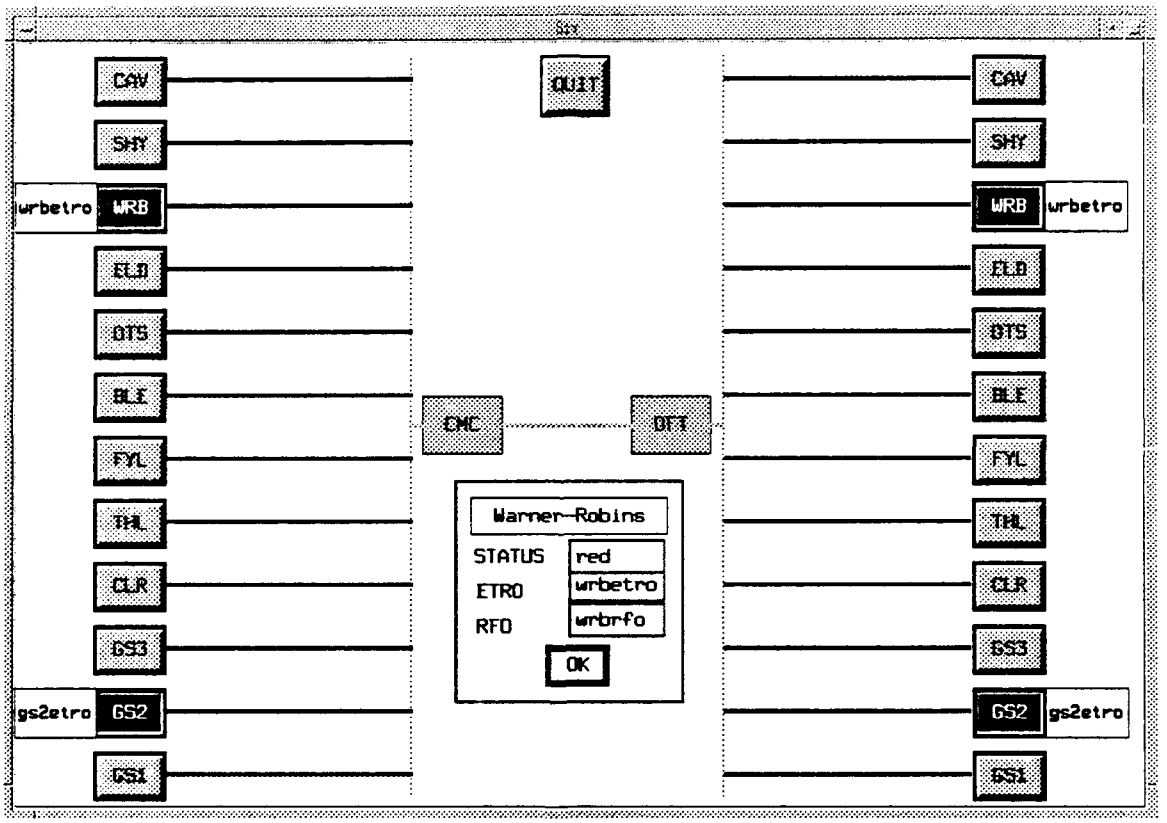


Figure 2-4 Spider Chart Display

The columns of rectangular boxes on the right and left sides of the spider chart display (for example, GS1, GS2) represent sensor sites. The rectangles in the middle of the display represent the correlation centers that collect information from the sensors. Each sensor site communicates with both correlation centers; this is represented by the duplication of sensor site boxes on both the right and left sides of the display. The lines represent communication lines between the sensor sites and the correlation centers. The status of sensors is represented by the shading of the rectangles. On a color display, the status would be represented using different background colors.

An operator may display detailed information concerning a sensor site by selecting a sensor site box corresponding to that sensor. This causes a detailed window to appear, displaying the status of the sensor, the date and time of the last message, the reason for outage (RFO) and the estimated time to returned operation (ETRO). These fields may be modified by the operator. Sensors may be in one of three states: operational, impaired, or down. For sensors that are not fully operational (i.e., the status is yellow) the ETRO is displayed to the outside of the sensor site box. ETROs are also displayed over communication lines that are not fully operational. The operator may also dynamically reconfigure the network[1] by adding/ deleting sensors to/from the network.

[1]The capability of dynamically reconfiguring the network does not exist in the spider chart example distributed with Serpent Version 1.0.

3 Specifying the Contract

The first step in creating a software system using Serpent is to apportion system functionality between the dialogue and the application. This involves creating a contract between the two components: defining the type and structure of information to be communicated, or shared, between the two components; establishing the range of values of this data; and establishing runtime communication between the components.

3.1 Defining Shared Data

Shared data is information that is communicated or shared between the application and dialogue. Defining shared data involves two steps:

1. Create the shared data definition file.
2. Run the created file through the Saddle processor.

The following is a brief description of each of these two steps. The *Serpent: Saddle User's Guide* contains a more complete description of both these steps.

Step 1: Create the shared data definition file. The shared data definition file defines the type and structure of information that can be shared between the application and dialogue. The shared data definition is specified in Saddle. By convention, the file is given the name of the application, followed by the extension `.sdd`.

Example 3-1 is an example of a shared data definition file for the spider application. The content of the shared data definition file is independent of the implementation language used. Note that these shared data record templates contain only information to define the application objects; they do not specify how the information is presented to the end user.

```
<< spiderA >>

spider: shared data

 sensor_sdd: record
    site_abbr: string[3];
    status: integer;
    site: string[32];
    last_message: string[8];
    rfo: buffer[32];
    etro: string[8];
 end record;

 cc_sdd: record
    name: string[3];
    status: integer;
 end record;

 communication line_sdd: record
```

```
    from_sensor: id of sensor_sdd;
    to_cc: id of cc_sdd;
      etro: string[8];
      status: integer;
 end record;

end shared data;
```

Example 3-1 Spider Shared Data Definition File

The file shown in Example 3-1 contains definitions for the data shared between the application and the dialogue for the spider application. The first line of the file contains the name (and possible path information) of the executable image of the application. This application is automatically executed by the Serpent command at runtime. (*Serpent: System Guide* contains a complete explanation of this process.) The three shared data record templates define the type and structure of the sensor, correlation center, and communication line application objects.

Step 2: Run the created file through the Saddle processor. Once the shared data has been defined in the file, it can be processed by Saddle to generate a C language header file. This file will have the same name as the shared data definition file with a different extension. For example, the shared data file `spiderA.sdd` will generate the header file `spiderA.h`. This header file can then be included in the C application and used to declare local variables of the shared data types. The C header file generated by running the shared data definition file shown in Example 3-1 through the Saddle processor is illustrated in Example 3-2.

```
#define MAIL_BOX "sss_mailbox"
#define ILL_FILE "sss.ill"

 typdef struct {
    char site_abbr[4];
    int status;
    char site[33];
    char last_message[9];
    char rfo[33];
    char etro[9];
 } sensor_sdd;

 typedef struct {
   char name[4];
   int status;
 } cc_sdd;

 typedef struct {
    id_type from_sensor;          /*ID of sensor */
    id_type to_cc;                /*ID of correlation center */
    char etro[9];
    int status;
 } line_sdd;
```

Example 3-2 C Language Header File

In Example 3-2, the first two lines in the file define two well-known constants: `MAIL_BOX` and `ILL_FILE`. These constants will be used in initializing Serpent. The three structures correspond to the record templates defined within the shared data definition file.

3.2 Data Types and Values

One output of processing the shared data definition file through the Saddle processor is a C header file containing corresponding C structures for the shared data records. These C structures can be used to declare local variables that correspond in size and structure to shared data records or used as casts. Components of shared data records can be declared as any of the following types: boolean, integer, real, string, ID or buffer. The C structures generated from these declarations depend on the type of the components. Example 3-3 is unrelated to the spider example used throughout this guide but includes a description of a shared data record that contains an example of each type of component.

```
employee_sdd: record
 name: string[32];
 salary: integer;
 exempt: boolean;
 experience: real;
 job_desc: buffer;
 self: id of employee_sdd;
end record;
```

Example 3-3 Shared Data Definition

Example 3-4 shows the C structure that is generated when the `employee_sdd` record is processed by Saddle processor.

```
typedef struct {
 char name[33];
 int salary;
boolean exempt;
double experience;
 buffer job_desc;
 id_type self;
} employee_sdd;
```

Example 3-4 Generated C Structure

Although each shared data component is now represented using a C language specific type, there is still a Serpent data type associated with each of them. The Serpent data type can be determined at runtime using the `get_shared_data_type` function illustrated in Example 3-5. The `serpent_data_type` is an enumeration of the different Serpent data types and is defined in Appendix A.

```
serpent_data_type type;
/*
** Get the Serpent type of the employee record salary
** component.
*/
 type = get_shared_data_type("employee", "salary");
```

Example 3-5 Serpent Data Type

Shared data values specified as strings in the shared data definition file are represented by character arrays in the C header files generated by the Saddle processor. It is therefore not necessary to allocate heap memory for these strings, although it is necessary to use the Unix `strcpy` function (see Example 3-6) or a similar utility to copy data into the character array.

```
/*
** Declare a local shared data variable.
*/
 employee_sdd employee;
/*
** Use strcpy to assign a string value.
*/
 strcpy (employee.name, "Harry Alter");
```

Example 3-6 Assigning Values to String Components

Shared data components of type integer, boolean, real, or ID can be assigned directly to C language variables. IDs are returned from a number of Serpent routines and are `id_type`. Saddle integers and booleans are actually of C type int and Saddle reals are actually of C type double. (See Example 3-7.)

```
#define false 0
#define true 1

/*
** Integer, boolean, real or id components can be set
** directly.
*/
 employee.salary = 45000;
 employee.exempt = false;
 employee.salary = 3.2;
```

Example 3-7 Assigning Values to Integer, Boolean, Real or ID Components

Buffer is the only dynamic shared data type in that neither the size nor the type of the information is predefined. Example 3-8 describes the buffer structure. Buffer type is required and specifies the type of information stored in the buffer. Buffer length is the size in bytes of the data and is required even if the data is of a well known type (i.e., integer). Buffer body is a pointer to the actual data. The space used to maintain this data is not part of the buffer structure and must be managed by the user.

```
typedef struct {
 serpent_data_types type;
 int length;
 caddr_t body;
} buffer;
```

Example 3-8 Buffer Structure

Buffers can be used to:

- Share untyped, contiguous data.
- Share large amounts of contiguous data (i.e., large strings).
- Provide variant records.

Example 3-9 contains the example of the `employee.site` buffer being used as a string.

```
/*
** This buffer is being used as a string.
*/
 employee.job_desc.type = serpent_string;
 employee.job_desc.length = strlen("Look busy")+ 1;
 employee.job_desc.body = malloc(employee.job_desc.length);
 strcpy(employee.job_desc.body, "Look busy");
```

Example 3-9 Assigning Values to Buffer Components

Shared data values can also be undefined. All uninitialized components of a shared data record instance created using the `add_shared_data` function are initialized by Serpent to be undefined. On the other hand, components of a local, shared data variable have whatever values are left by the system—most likely zeros. If this structure is used to initialize the shared data instance (with the `add_shared_data` or `put_shared_data` routines), all the components of the instance are initialized with these values. Components of local, shared data variables can be explicitly set to undefined using the `set_undefined` routine illustrated in Example 3-10. The `is_undefined` function can be used to determine if a component value is undefined.

```
/*
** The set_undefined function is used to set the value of
** a component to undefined.
*/
 set_undefined(serpent_buffer, &employee.job_desc);
```

Example 3-10 Setting Component Values to Undefined

3.3 Initialization and Cleanup

The first task of any Serpent application is to initialize the system. Serpent initialization establishes communication between the application and the dialogue. The final application task is to clean up the Serpent system environment before exiting. The code segment from the spider application shown in Example 3-11 illustrates the basic operations necessary for Serpent initialization and cleanup.

```
#include "serpent.h"   /* serpent interface definition */

main() {
 serpent_init(MAIL_BOX,ILL_FILE);
 serpent_cleanup();
}
```

Example 3-11 Serpent Initialization

Specification Steps:

1. ***Include Serpent header file.*** The `serpent.h` file contains the external definition for the Serpent interface.
2. ***Initialize Serpent.*** The `serpent_init` procedure is used to initialize Serpent. It takes as parameters the `MAIL_BOX` and `ILL_FILE` constants generated by the Saddle processor. This procedure establishes communication between the application and the dialogue manager.
3. ***Clean up.*** The `serpent_cleanup` routine must be invoked before exiting the application. It is important to complete this step to release allocated system resources.

4 Modifying Information

The application can add, change, or remove information to and from the shared database using the transaction mechanism described in the introductory chapter of this document. Together, these are considered modifications to the shared database. The collection of application data in the shared database is known as the *view*. This is the information that is available to the dialogue writer to be presented to the end user. The view can be modified by either the application or the dialogue.

4.1 Sending Transactions

Before information can be modified in the shared database, it is necessary to start a transaction. All modifications to the shared database must be performed as part of a transaction.

It is possible to have multiple transactions open at one time. Each transaction has a unique transaction handle. Every operation performed on or to a transaction must specify this transaction handle.

The actual change to the shared database does not occur until the transaction is committed. Up to this point it is also possible to roll back the transaction so that none of the changes to shared data occur.

The code segment from the spider application in Example 4-1 shows the operations necessary for sending transactions. Code and comments directly related to the task are emphasized in bold type.

```
#include "serpent.h"   /* serpent interface definition */

main() {
 transaction_type transaction; /* transaction handle */

 serpent_init(MAIL_BOX,ILL_FILE);
 transaction = start_transaction();
 commit_transaction(transaction);
 serpent_cleanup();
}
```

Example 4-1 Sending Transactions

Specification Steps:

1. Declare ***transaction variable.*** A local variable of `transaction_type` can be used to maintain a transaction handle.

2. ***Start a transaction.*** The `start_transaction` function returns a transaction handle that must be passed to any subsequent commands operating on the transaction.

3. ***Commit the transaction.*** The actual change to shared data does not occur until the transaction is committed. Up to this point it is also possible to roll back the transaction using the `rollback_transaction` routine so that none of the changes to shared data occur.

4.2 Adding Static Information

This section makes some simplifying assumptions about the application that may in fact hold true for simple programs. The primary assumption is that the application will create only a fixed number of shared data instances so that the IDs of these instances can be maintained in local variables. A secondary assumption is that the application will create no more than one instance of each shared data element.

At any given moment, there can be up to three different versions of any given shared data instance. First, there is a local copy in the application. Second, there can be a copy that is part of an open transaction. Third, there is a copy in the shared database. Depending upon whether the shared data instance has been last modified by the application or by the end-user, the more current copy could be either the local application or shared database copy. A shared data instance that is part of an open transaction is the delta from the more current to less current copy of the shared data instance. The shared data copy being affected by any given operation should be apparent from the context.

Variables of generated shared data types are referred to as shared data variables. The first step in adding information to shared data is to assign values to these shared data variables. The method for doing this is based on the Serpent types of the components and is explained in detail in Section 3.2. These variables can then be used to initialize a record instance, either a component at a time or the entire record at once.

Once a transaction has been started, you can begin to add, change or remove information to/from the shared database as part of this transaction. These changes are made as part of the transaction and are not applied to the shared database until the transaction is committed.

The code segment from the spider application in Example 4-2 illustrates the operations involved in adding information to the shared database. Code and comments directly related to the task are emphasized in bold type.

```
#include "serpent.h"   /* serpent interface definition */
#include "spiderA.h"   /* application data structures */
#define GREEN_STATUS 0
#define YELLOW_STATUS 1
#define RED_STATUS 2

main() {
 transaction_type transaction;  /* transaction handle */
 cc_sdd cmc;                    /* shared data variables */
 sensor_sdd gs1;                /* shared data variables */
 id_type cmc_id,gs1_id;         /* object instances */

 serpent_init(MAIL_BOX,ILL_FILE);
/*
** Initialize shared data variables.
*/
 strcpy(cmc.name, "CMC");
 cmc.status = GREEN_STATUS;

 gs1.status = RED_STATUS;
/*
** Start a transaction to be sent to the dialogue.
*/
 transaction = start_transaction();
/*
** Create an instance of the correlation center shared data
** record in the transaction and initialize using the shared
** data variable.
*/
 cmc_id = add_shared_data(
    transaction,"correlation_center", NULL, &cmc
 );
/*
** Create an instance of the sensor shared data record but
** this time update only the name component.
*/
 gs1_id = add_shared_data(
   transaction,"sensor", "name", &gs1.name
 );

 commit_transaction(transaction);

 serpent_cleanup();
}
```

Example 4-2 Adding Information to the Shared Database

Specification Steps:

1. ***Include Saddle generated header file.*** This file (spiderA.h in the example) defines the structure of the shared data. The file serpent.h must be included before spiderA.h because spidera.h uses types defined in serpent.h.

2. ***Define constants.*** The spider example contains three constants: `GREEN_STATUS`, `YELLOW_STATUS`, and `RED_STATUS`. These constants are not required but help increase the clarity of the example.

3. ***Define shared data variables.*** Variables `cmc` and `gs1` are both instances of generated shared data structures. These variables are used to initialize instances of shared data in the shared database.

   The variables `cmc_id` and `gs1_id` are used to store the ids of the created shared data instances. These variables are declared to be of `id_type`. The ids are necessary to perform further operations on these instances in the shared database.

4. ***Assign values to shared data variables.*** The mechanism for accomplishing this task depends on the component types. This is explained in detail in Section 3.2.

5. ***Add information to the shared database.*** The `add_shared_data` routine creates a shared data instance as part of the specified transaction and returns the ID of the instance. The routine allows you to initialize a single component of the instance by specifying the name of the component and providing a *pointer* to the initial value. Any uninitialized fields of the instance are left undefined. It is also possible to initialize the entire instance by providing a pointer to the structure and specifying NULL for the component name.

4.3 Adding Dynamic Information

In larger, more complex software systems it is not always practical to declare a fixed collection of variables to represent data. The management of dynamically created data structures is often radically different from that of static variables. The purpose of this section is to reexamine how the application can create and manage shared data in this more complex environment. To do this, the spider chart example has been modified to display *n* sensors, where *n* is defined in an external file along with the sensor data.

The code segment from the spider application in Example 4-3 illustrates the operations involved in adding dynamic information to the shared database. Code and comments directly related to the task are emphasized in bold type.

```
#include "serpent.h"    /* serpent interface definition */
#include "spiderA.h"    /* application data structures */

main() {
 transaction_type transaction;
 HASH id_table;/* hash of id's of all shared data in view */
 sensor_sdd *sensor;

 FILE *fp;              /* file pointer to the data file    */
 int i = 0;             /* counter used for loading sensors */
 int sensor_count;      /* number of sensors                */
```

```
 serpent_init(MAIL_BOX,ILL_FILE);
/*
** Create an id hashtable. This table contains the ids of
** all the shared data instances in the shared database at
** any given time.
*/
 id_table = make_hashtable(
    MAX_HASH, sss_hash_id, sss_match_id
 );
/*
** Start a transaction.
*/
 transaction = start_transaction();
/*
** Sensor data is stored in an external file.
*/
 fp = fopen("sdata", "r");
/*
** The first field in the file contains the number of sensors.
*/
 fscanf(fp, "%d", &sensor_count);
/*
** Read in each sensor and put into shared data.
*/
 i = 0;
 while (i++ < sensor_count) {/* while sensor's left    */
/*
** Create shared data record instance.
*/
    sensor = (sensor_sdd *)make_node(sizeof(sensor_sdd));
/*
** Read in sensor data into shared data record instance.
** Note: '&' address operators are omitted from most
** componerts because they are arrays and are already passed
** by reference.
*/
    fscanf(
      fp, "%s%d%s%s%s%s",
      sensor->site_abbr,
      &sensor->status,
      sensor->site,
      sensor->last_message,
      sensor->rfo,
      sensor->etro
    );
/*
** Put sensor into shared data.
*/
    sensor->self = add_shared_data(
      transaction, "sensor_sdd", NULL, NULL
   );

    put_shared_data(
      transaction,sensor->self, "sensor_sdd", NULL, sensor
   );
/*
** Add shared data instance to id table.
*/
    add_to_hashtable(id_table, sensor, sensor->self);

 }            /* end while file not empty                */
/*
```

```
** Close the data file.
*/
 fclose(fp);
 commit_transaction(transaction);
 serpent_cleanup();
}
```

Example 4-3 Adding Information to the Shared Database

Specification Steps:

1. ***Add self component to shared data record templates.*** The `self` component is used to store the ID of a shared data instance directly in the shared data structure. This field of the structure is used in the example to identify the particular shared data instance. The self component is of type `id`. To add this component to the shared data record template, it is necessary to modify the shared data definition file and then repeat the steps described in Section 3.1.

2. ***Include Saddle generated header file.*** The spiderA.h header file contains the shared data structures for the spider example. This file must be included after serpent.h since serpent_A.h uses types defined in this other file.

3. ***Define local variables.*** The next preliminary step is to define local variables. The `sensor` variable declared in Example 4-4 is no longer an instance but a pointer to the `sensor_sdd` structure. This variable will be used to reference dynamically allocated local instances of the sensor structure.

   The `id_table` variable is a hash table that is used to store and access local instances of sensor data. The hash table has proven to be a very effective data structure for accomplishing this since the lookup can be done using ID values, which are unique. An implementation of hash tables, as well as other abstract data structures, are included in the C-programmer's toolkit that is distributed with Serpent.

4. ***Create local data structures.*** A local shared data instance is created (allocated from memory) for each sensor in the file. These instances are stored in a hash table that must also be created.

5. ***Initialize local shared data instances.*** Each local shared data value must then be initialized. In Example 4-4, this is accomplished by reading the data directly into the `sensor` structure from the file.

6. ***Add information to the shared database.*** Once a transaction has been started and the data initialized, information can be added to the shared database. In Example 4-4, this is a two-step procedure. An uninitialized sensor instance is created with the `add_shared_data` function. The resulting id is assigned to the `self` component of the local shared data instance. The sensor instance is then initialized using the `put_shared_data` procedure that behaves much like the `add_shared_data` function but operates on an existing shared data instance.
7. ***Update the local database.*** The last step is to add the local shared data instance to the hash table using the ID of the shared data instance. This ID can later be used as an identifier in updating changes to shared data in local data.

4.4 Modifying Information

Shared data instances in transactions or in the shared database can be modified using the `put_shared_data` procedure. This procedure takes as a parameter the ID of the shared data instance.

It is possible to modify any single component of a shared data record instance, or the entire record. Unmodified components in the transaction are marked as unchanged and maintain their current values. This is different from components that are explicitly set to undefined, which is actually a value.

The code segment from the spider application in Example 4-4 illustrates the operations involved in adding dynamic information to the shared database. Code and comments directly related to the task are emphasized in bold type.

```
#include "serpent.h"   /* serpent interface definition */
#include "spiderA.h"   /* application data structures */

main() {
 transaction_type transaction;
 sensor_sdd gs1;                /* shared data variables */
 id_type cmc_id,gs1_id;         /* object instances */

 serpent_init(MAIL_BOX,ILL_FILE);
 transaction = start_transaction();
/*
** Update the name component of the sensor using a
** string constant.
*/
 put_shared_data(
    transaction, gs1_id, "sensor", "status", "GS1"
```

```
 );
 commit_transaction(transaction);
 serpent_cleanup();
}
```

Example 4-4 Modifying Information in the Shared Database

Specification Task

Modifying information in the shared database. The `put_shared_data` routine modifies the values of shared data instances that have already been created and are part of a transaction. This routine works in an identical manner to the `add_shared_data` call except that it takes an extra parameter, the ID of the shared data instance to be modified. The `put_shared_data` routine in Example 4-5 is used to assign a value (a string) to the name component of the first shared data instance.

4.5 Removing Information

Shared data instances in transactions or in the shared database can be removed using the `remove_shared_data` procedure. It is not possible to remove components of shared data record instances.

The code segment from the spider application in Example 4-5 illustrates the operations involved in removing information from the shared database. Code and comments directly related to the task are emphasized in bold type.

```
#include "serpent.h"   /* serpent interface definition */
#include "spiderA.h"   /* application data structures */

main() {
 transaction_type transaction;
 sensor_sdd gsl;                /* shared data variables */
 id_type cmc_id,gsl_id;         /* object instances */

 serpent_init(MAIL_BOX,ILL_FILE);
 transaction = start_transaction();
/*
** Update the name component of the sensor using a
** string constant.
*/
 remove_shared_data(transaction, "sensor_sdd", gsl_id);
 commit_transaction(transaction);
 serpent_cleanup();
}
```

Example 4-5 Removing Information from the Shared Database

Specification Task

Removing information from the shared database. The `remove_shared_data` procedure is used to remove a shared data instance from either the transaction or the shared database. The procedure takes a transaction handle, the element name, and the ID of the shared data instance to be deleted as parameters.

5 Retrieving Information

Serpent implements an *active database model* from the perspective of the application interface. This means that changes to application data resulting from end-user interactions with the system are automatically communicated back to the application, using the same transaction mechanism described in Section 4.3.

Transactions from the dialogue to the application consist of a list of changed shared data instances. The following assumptions are true about incoming transactions:

- Incoming transactions are guaranteed to have at least one changed shared data instance since empty transactions are automatically discarded by the interface.
- Changed shared data elements appear in random order in the transaction.
- Transactions remain unmodified in memory until the transaction is purged. This allows the application developer, for example, to reexamine changed instances.

5.1 Retrieving Transactions

The code segment from the spider application shown in Example 5-3 illustrates the basic operations of retrieving information from the shared database.

Specification Steps:

1. ***Get the transaction.*** The Serpent interface provides both synchronous and asynchronous calls for getting information from the shared database. The `get_transaction` routine waits until a transaction is available and then returns a handle for this transaction. The `get_transaction_no_wait` routine returns `not_available` when no transaction is available.
2. ***Get each changed shared data instance.*** The `get_first_changed_element` routine returns the first changed shared data element instance in the transaction and marks it as the *current* element. The `get_next_changed_element` routine returns the element directly following the current element and marks it as current. The `null_id` is returned if there is no next element instance on the list.
3. ***Purge the transaction.*** Once the transaction has been fully processed, it should be purged from the system. This frees system resources that could otherwise run out.

Code and comments directly related to the task are emphasized in bold type.

```
main() {
 boolean done = false;
 id_type id;
 transaction_type transaction;

 serpent_init(MAIL_BOX,ILL_FILE);
                :
/*
** Retrieve information from shared database.
*/
 while (!done) {

    transaction = get_transaction()

    id = get_first_changed_element(transaction);
/*
** Get each changed instance in the transaction.
*/
    while (id != null_id) {
       id = get_next_changed_element(transaction);
   }

    purge_transaction(transaction);

 }

 return();
}
```

Example 5-1 Transaction Processing

5.2 Incorporating Changes

Changed element instances from the dialogue need to be processed for any changes in the application domain to be affected. The Serpent application interface provides several routines for the purpose of processing changed shared data elements.

This section makes some simplifying assumptions about the application that may in fact hold true for simple programs. The primary assumption is that the application has created only a fixed number of shared data instances so that the IDs of these instances can be maintained as static, local variables. A secondary assumption is that the application has created no more than one instance of each shared data record.

The code segment from the spider application in Example 5-2 illustrates the operations involved in incorporating changes to shared data elements in static, local variables. Code and comments directly related to the task are emphasized in bold type.

```
main() {

 id_type id;
 transaction_type transaction;
 string element_name;
                :
```

```
    id = get_first_changed_element(transaction);
/*
** Get each changed record instance in the transaction.
*/
    while (id != null_id) {

      element_name = get_element_name(transaction, id);
/*
** If the record is a correlation center then this must
** be the cmc shared data variable.
*/
      if (strcmp(element_name, "cc_sdd") == 0) {
        incorporate_changes(transaction, id, &cmc);
      }
/*
** Otherwise, this must be the gsl variable.
*/
     else {
        incorporate_changes(transaction, id, &gsl);
      }

      id = get_next_changed_element(transaction);
    }
              :
 return();
}
```

Example 5-2 Processing Changes to Shared Data Records (Simple Programs)

Specification Steps:

1. ***Get the element name.*** This is a simple call that returns a pointer to the element name. For simple programs that have no more than one instance of a particular shared data record, the element name can be used to identify the shared data instance. In larger, more complex systems it is often useful in determining a class of shared data instances.
2. ***Update local database.*** Shared data variables can be updated using the `incorporate_changes` routine. This routine directly incorporates changes in the shared data instance into the local variable. Components of the shared data record that have not been changed are left untouched. By continually incorporating changes into the initial shared data variable, the application developer is guaranteed that application data remains consistent with user input.

5.3 Processing Dynamic Elements

In larger systems, it is often necessary to dynamically create shared data records, or instantiate multiple instances of a single shared data record. This section describes how changed shared data record instances can be managed in these situations.

In the dialogue, it is also possible to create, modify, or remove instances of application shared data as a result of input from the end user. This is known as the *change type*. Each shared data record instance in a transaction has an associated change type.

These steps are illustrated in Example 5-3 taken from the spider chart example. Code and comments directly related to the task are emphasized in bold type.

```
main() {

 id_type id;
 transaction_type transaction;
 sensor_sdd *sensor;

 change_type change;
                :
    id = get_first_changed_element(transaction);
/*
** Get each changed record instance in the transaction.
*/
    while (id != null_id) {

       change = get_change_type(transaction, id);
/*
** Update the local database based on the change type.
*/
      switch(change) {

        case create:
            sensor = get_shared_data(transaction, id, NULL);
            add_to_hashtable(id_table, sensor, id);
          break;

        case modify:
            sensor = get_from_hashtable(id_table, id);
            incorporate_changes(transaction, id, sensor);
          break;

        case remove:
            sensor = delete_from_hashtable(id_table, id);
          free(sensor);
          break;
      }                 /* end switch                */

       id = get_next_changed_element(transaction);
    }
                :
 return();
}
```

Example 5-3 Processing Changes to Shared Data Records (Large Systems)

Specification Steps:

1. ***Determine the change type.*** The change type can be obtained using the `get_change_type` function.

2. ***Update the local database based on the change type.*** The exact type of processing required to update the local database is based primarily on the change type. If this is a new shared data element (e.g., the change type is `create`) the `get_shared_data` function can be used to create a copy of the record instance. The memory required for this copy is automatically allocated from process memory using the `make_node` function provided in the C Toolkit memoryPack utility distributed with Serpent. This copy can then be added to the hash table.

If the change type is `modify`, the local shared data instance can be obtained from the hash table. The `incorporate_changes` routine can then be used to update the contents of this instance with changed component values.

If the shared data record instance in the shared database has been deleted, the application developer may want to delete it from the local database as well. Memory allocated by the `get_shared_data` function should be freed using the `free_node` procedure provided in the C Toolkit memoryPack utility.

5.4 Examining Changes by Component

The Serpent application programmer's interface provides routines that allow the application developer to examine each changed component in a changed record individually.

The operations are illustrated in Example 5-4, taken from the spider chart example. Code and comments directly related to the task are emphasized in bold type.

```
main() {

 id_type id;
 transaction_type transaction;
 sensor_sdd sensor;
 string element_name;

 serpent_data_types type;
 id_type *id_data;
 string component_name;
 LIST changed_components;
                :
    id = get_first_changed_element(transaction);
/*
** Get each changed record instance recording the transaction.
*/
    while (id != null_id) {

       changed_components = create_changed_component_list(
        transaction,id
     );
```

```
    loop_through_list(
      changed_components, component_name, string
) {

      type = get_shared_data_type(
        element_name, component_name
      );
      if (type == serpent_id) {
         id_data = (id_type *)get_shared_data(
           transaction, id, component_name
       );
      }                    /* endif id type            */

   }                       /* end loop through list    */

      free_list(changed_components, NOT_NODES);

      id = get_next_changed_element(transaction);
   }
            :
return();
}
```

Example 5-4 Processing Changes to Shared Data Records (Large Systems)

Specification Steps:

1. ***Get the list of changed components.*** A list of changed components can be obtained by using the `create_changed_component_list` function.
2. ***Loop through the list.*** The listPack contained in the C toolkit distributed with Serpent provides a number of routines that can be used for processing the elements on a list.
3. ***Examine the type and/or data.*** The Serpent application programmer's interface provides routines to examine both the type and the data at the component level. The `get_shared_data_type` returns a `serpent_data_type`. The `get_shared_data` routine can also be used to return the component value instead of the element instance value by specifying the name of a component. Serpent automatically allocates the storage for the component value but it is the application programmer's responsibility to free this storage (using the `free_node` procedure available in the C toolkit memoryPack).
4. ***Destroy the changed component list.*** The changed component list can be destroyed using the `free_list` routine from the C toolkit listPack.

6 Finishing the Application

Other than sending and retrieving data, the application can determine errors from the use of Serpent, record communication between the application and Serpent and exit according to a signal received from the dialogue.

6.1 Error Checking

Each routine in Serpent sets status on exit. It is good software engineering practice to check status after every call to make sure that the routine has executed correctly, and provide appropriate recovery actions if it has not. Example 6-1 illustrates the routines provided by Serpent for examining the status.

```
transaction = start_transaction();
if(get_status() != ok) {
 print_status("error during start_transaction");
 serpent_cleanup();
}
```

Example 6-1 Examining Status

The first of these routines is `get_status`, which returns an enumeration of status codes. Valid statuses returned by each routine in Serpent are defined in Appendix B. Successful execution (or "OK") is always set to zero; hence, it is possible to make a simple boolean comparison for bad status.

The `print_status` routine prints a user-defined error message and the current status.

6.2 Recording Transactions

Transactions between the application and the dialogue can be recorded using the `start_recording` and `stop_recording` procedures available in the Serpent application programmers interface. After the call to `start_recording` is made, transactions may be sent across the interface. Any number of transactions containing any type or amount of data can be sent. Once `start_recording` has been called, all transactions and associated data will be written to the specified file until the `stop_recording` routine is invoked.

Transactions can be examined using the `format` command described in Section 7.1. This is useful in debugging since it allows the examination of information flow across the interface. Transactions can also be played back to simulate either application or dialogue functionality using the `playback` command described in Section 7.2.

Before testing the application or the dialogue, first record the transactions to be used in testing. Example 6-2 illustrates the basic operations for recording transactions.

```
 transaction tvpe transaction;
                :
/*
** Start recording.
*/
 start_recording("recording", "test data: 5.7.3");
/*
** Send test data.
*/
 transaction = start_transaction();
                :
 commit_transaction(transaction);

 transaction = start_transaction();
                :
 commit_transaction(transaction);

 transaction = start_transaction();
                :
 commit_transaction(transaction);

/*
** Stop recording.
*/
 stop_recording();
                :
}
```

Example 6-2 Recording Transactions

Specification Steps:

1. ***Start recording.*** The start_recording routine takes as parameters both the name of the file in which to save the recording and a message to help identify the recording.
2. ***Send transactions.*** After the call to start_recording is made, transactions may be sent across the interface.
3. ***Stop recording.*** The stop_recording function closes the current recording file.

6.3 Dialogue Initiated Exit

The dialogue can terminate at any time using the exit command available to the dialogue specifier. The exit command sends a SIGINT signal to the application. This signal will cause the application to exit immediately, unless a signal handler has been registered with the operating system.

The signal handler describes the steps to be taken when the dialogue initiates an exit. Typically, this involves saving data structures out to permanent storage and exiting the system.

The code segment in Example 6-3 taken from the spider chart illustrates the operations necessary to handle dialogue initiated exit. Code and comments directly related to the task are emphasized in bold type.

```
#include <signal.h>                    /* UNIX signal handling */

void spider_signal_handler()
{
 serpent_cleanup();
 exit (0);
}

main ()
{
 signal(SIGINT, spider_signal_handler);

 serpent_init(MAIL_BOX, ILL_FILE);
                :
 serpent_cleanup();
}
```

Example 6-3 Signal Handler for Dialogue Initiated Exit

Specification Steps:

1. ***Include Unix signal handling.*** The `signal.h` include file contains the external definitions for signal processing.
2. ***Write the signal handler.*** The signal handler describes the steps to be taken when the dialogue initiates an exit. Remember to invoke `serpent_cleanup` before exiting.
3. ***Register the signal handler.*** The signal handler must be associated with the `SIGINT` signal. This is accomplished using the Unix `signal` function.

7 Testing and Debugging

The recording capability discussed in Chapter 3 provides a mechanism to assist in testing and debugging.

7.1 Formatting Recordings

Application recordings are saved in a binary format file. The `format` command distributed with Serpent converts this file into a formatted, easy-to-read report. The information in the file can be useful in isolating problems to either the application or the dialogue.

```
% format recording
FORMATTING JOURNAL FILE: recording

HEADER:
   dialogue name:
   message: no comment at this time
OWNER:
   ill file name: se.ill
   mailbox name: SE_BOX

PARTICIPANT:
   ill file name: se.ill
   mailbox name: DM_BOX

TRANSACTION:
 Fri Jan 25 15:17:13.800 1991
   Sender: SE_BOX
   Receiver: DM_BOX
   Element name: dialogue_sdd Change type: create ID: 955
      shared_data            buffer     UNDEFINED_BUFFER
      termination            buffer     UNDEFINED_BUFFER
      macros                 buffer     UNDEFINED_BUFFER
      externs                buffer     UNDEFINED_BUFFER
      initialization         buffer     UNDEFINED_BUFFER
      count                  integer    0
      name                   string     UNDEFINED_STRING
      prologue               buffer     UNDEFINED_BUFFER
%
```

Example 7-1 Formatting the Recording File

7.2 Playback

Once you have made a recording, it is possible play back the recording to simulate one or more of the Serpent processes. To simulate the spider application, for example, you would run the `playback` command provided with Serpent specifying the name of the recording file and the mailbox of the process to be simulated, as illustrated in Example 7-2.

```
% app-test recording SPIDERA_BOX
Playing back journal file: recording
Message: regression test data, 5.7.3
Playback completed successfully
% _
```

Example 7-2 Testing the Application

Appendix A Data Structures

This appendix presents in alphabetical order the type and constant definitions that are used in the C language interface to the Serpent system. The following is a list and short description of each of these types and constants. A more complete description immediately follows:

Type/Constant	Description
`buffer`	used to define the structure of a shared data buffer
`change_type`	defines the type of modification made for an element
`id_type`	used to uniquely identify shared data elements
`null_id`	defines the null value for the `id_type`
`serpent_data_types`	an enumeration of defined Serpent data types
`transaction_type`	used to define transaction handles

TYPE

buffer

Description

The `buffer` type allows the communication of *n* bytes of application data along with an indication of the type. `Buffer` is the only dynamic shared data type in that neither the size nor the type of the information is predefined. Buffers can be used to: share untyped, contiguous data; share large amounts of contiguous data (i.e., large strings); provide variant records.

Definition

```
typedef struct {
 int length;
 caddr_t body;
 serpent_data_type type;
} buffer;
```

Components

`length`	Size in bytes of the data. This field is required even if the data is of a well known type (i.e., integer).
`body`	A pointer to the actual data. The space used to maintain this data is not part of the buffer structure and must be managed by the user.
`type`	The type of information stored in the buffer. This field is also required.

TYPE

change_type

Description

The `change_type` defines the type of modification made for an element.

Definition

```
typedef enum change_type {
 no_change = -1,
 create = 0,
 modify = 1,
 remove = 2,
 get = 3
} change_type;
```

Components

`no_change`	Not changed or invalid change.
`remove`	Existing shared data instance removed.
`create`	New shared data instance created.
`modify`	Existing shared data instance modified.
`remove`	Existing shared data instance removed.

TYPE

id_type

Description	The `id_type` is used to uniquely identify shared data elements.
Definition	`typedef private id_type;`

CONSTANT

null_id

Description	The `null_id` constant defines the null value for the `id_type`. This constant can be used to test for null ID values.
Definition	`#define null_id(iid_id_type)-1`

TYPE

serpent_data_type

Description

The `serpent_data_type` type is an enumeration of defined Serpent data types.

Definition

```
typedef enum data_type {
 serpent_null_data_type =-1,
 serpent_boolean =0,
 serpent_integer =1,
 serpent_real =2,
 serpent_string =3,
 serpent_record =4,
 serpent_id =5,
 serpent_buffer =6,
 serpent_undefined =7
} serpent_data_type;
```

TYPE

transaction_type

Description — Variables of `transaction_type` are used to define transactions.

Appendix B Routines

This appendix presents in alphabetical order the functions and procedures that make up the C language interface to Serpent. These routines can be categorized as follows:

Initialization/Cleanup

- serpent_init
- serpent_cleanup

Transaction Processing

- start_transaction
- commit_transaction
- rollback_transaction
- get_transaction
- get_transaction_no_wait
- purge_transaction

Sending and Retrieving Data

- add_shared_data
- put_shared_data
- remove_shared_data
- get_first_changed_element
- get_next_changed_element
- get_shared_data
- incorporate_changes
- create_changed_component_list
- get_change_type
- get_element_name
- get_shared_data_type

Undefined Values

- set_undefined
- is_undefined

Record/Playback

- start_recording
- stop_recording

Checking Status

- get_status
- print_status

FUNCTION

add_shared_data

Description

The `add_shared_data` routine creates an instance for the specified shared data element and returns a unique ID. The shared data instance may or may not be initialized.

Syntax

```
id_type add_shared_data(
 /* transaction : in transaction_type */
 /* element_name : in string */
 /* component_name : in string */
 /* data : in caddr_t */
);
```

Parameters

`transaction`	The transaction for which this operation is defined.
`element_name`	The name of the shared data element.
`component_name`	The name of a specific component to be initialized with the data, or null if the data corresponds to the entire element.
`data`	Data or null pointer if non-initialized.

Returns

The ID of the newly created shared data instance.

Status

`ok, out_of_memory, null_element_name, overflow`

ROUTINE

commit_transaction

Description	The `commit_transaction` procedure is used to commit a transaction to the shared database.

Syntax	`void commit_transaction(`
	`  /* transaction: in transaction_type */`
	`);`

Parameters	`transaction` Existing transaction ID.
Status	`ok, out_of_memory, invalid_transaction_handle`

FUNCTION

create_changed_component_list

Description	The `create_changed_component_list` function accepts an instance ID as a parameter and creates a list of changed component names. This component list is managed using the C Toolkit listPack distributed with Serpent.

Syntax	`LIST create_changed_component_list(`
	`  /* id: in id_type */`
	`);`

Parameters	`id` Existing data instance ID.
Returns	The list of changed component names associated with a data instance, or `NULL` if none.
Status	`ok, invalid_id, out_of_memory, element_not_a_record`

FUNCTION

get_change_type

Description	The `get_change_type` function accepts an instance ID as a parameter and returns the associated change type.

Syntax	`change_type get_change_type(`
	`  /* id : in id_type */`
	`);`

Parameters	`id` Existing shared data ID.
Returns	Element name associated with the shared data instance ID.
Status	`ok, invalid_change_type, invalid_transaction_handle,` invalid_id

FUNCTION

get_element_name

Description	The `get_element_name` function accepts an instance ID as a parameter and returns the associated element name.

Syntax	`string get_element_name(`
	`  /* id : in id_type */`
	`);`

Parameters	`id` Existing shared data ID.
Returns	Element name associated with the shared data instance ID.
Status	`ok, invalid_id`

FUNCTION

get_first_changed_element

Description	The `get_first_changed_element` function used to get the ID of the first changed element in a transaction.

Syntax	`id_type get_first_changed_element(`
	`  /* transaction_type : in transaction */`
	`);`

Parameters	`transaction` Existing transaction ID.
Returns	The handle of the first changed element.
Status	`ok, invalid_transaction_handle, out_of_memory`

FUNCTION

get_next_changed_element

Description	The `get_next_changed_element` function is used to get the ID of the next changed element on a transaction list or return `null_id` if the transaction list is empty.
Syntax	id_type get_next_changed_element(/* transaction_type : in transaction */);
Parameters	`transaction` Existing transaction ID.
Returns	The handle of the next changed element.
Status	`ok, invalid_transaction_handle, out_of_memory`

FUNCTION

get_shared_data

Description

The `get_shared_data` function allocates process memory, copies shared data into process memory, and returns a pointer to the data.

***Warning*: Record components may not have been specified and, therefore, would not contain valid data.**

Syntax

```
caddr_t get_shared_data(
 /* transaction : in transaction_type */
 /* id : in id_type */
 /* component_name : in string */
);
```

Parameters

`transaction`	Transaction in which to find the shared data ID.
`id`	Existing shared data ID.
`component_name`	Name of component for which to retrieve data, or entire element if NULL.

Returns

A pointer to changed data.

Status

`ok, invalid_id, out_of_memory, incomplete_record`

FUNCTION

get_shared_data_type

Description

The `get_shared_data_type` function is used to get the type associated with a shared data element.

Syntax

```
serpent_data_types get_shared_data_type(
 /* element_name: in string */
 /* component_name: in string */
);
```

Parameters

`element_name`	The name of the shared data element.
`component_name`	The name of the shared data component, or NULL.

Returns

The type of the shared data element or record component.

Status

`ok, null_element_name`

FUNCTION

get_status

Description	The `get_status` function returns the current system status.
Syntax	`isc_status get_status();`
Parameters	None.
Returns	The current status.
Status	None.

FUNCTION

get_transaction

Description	The `get_transaction` function is used to synchronously retrieve the ID for the next completed transaction.
Syntax	`transaction_type get_transaction();`
Parameters	None.
Returns	The transaction ID for a completed transaction.
Status	`ok, system_operation_failed`

FUNCTION

get_transaction_no_wait

Description	The `get_transaction` function is used to asynchronously retrieve the ID for the next completed transaction.
Syntax	`transaction_type get_transaction_no_wait();`
Parameters	None.
Returns	The transaction ID for a completed transaction.
Status	`ok, system_operation_failed, not_available`

PROCEDURE

incorporate_changes

Description	The `incorporate_changes` procedure is used to incorporate changes into local process memory without destroying unchanged information.

Syntax

```
void incorporate_changes(
 /* transaction : in transaction_type */
 /* id : in id_type */
 /* data : in out caddr_t */
);
```

Parameters		
	`transaction`	Existing transaction ID.
	`id`	Existing shared data ID.
	`data`	*Pointer to the local data structure* to be updated.

Status	`ok, invalid_id`

FUNCTION

is_undefined

Description

The `is_undefined` function evaluates a data value of a specified type and determines if the value is undefined. The `is_undefined` function cannot be used with an entire shared data record at once.

Syntax

```
boolean is_undefined(
 /* type : in serpent_data_type */
 /* data : in caddr_t */
);
```

Parameters

`type`	The type of the shared data component.
`data`	Pointer to the value being examined.

Returns

True if data is undefined; false otherwise.

Status

`ok, operation_undefined_type`

PROCEDURE

print_status

Description	The `print_status` procedure prints out a user-defined error message and the current status.

Syntax	`void print_status(`
	`  /* error_msg : in string */`
	`);`

Parameters	`error_msg` User-defined error message.
Status	None.

PROCEDURE

purge_transaction

Description	The `purge_transaction` procedure is used to remove a received transaction once the contents of the transaction have been examined and acted upon.

Syntax	`void purge_transaction(`
	`  /* transaction : in transaction_type */`
	`);`

Parameters	`transaction` Existing transaction ID.
Status	`ok, invalid_id, illegal_receiver`

PROCEDURE

put_shared_data

Description

The `put_shared_data` call is used to put information into shared data.

Syntax

```
void put_shared_data(
 /* transaction : in transaction_type */
 /* id : in id_type */
 /* element_name : in string */
 /* component_name : in string */
 /* data : in caddr_t */
);
```

Parameters

`transaction`	The transaction to which the shared data should be put.
`id`	Shared data ID.
`element_name`	The name of the shared data element.
`component_name`	The name of the shared data component.
`data`	Shared data.

Status

`ok, undefined_shared_data_type, null_element_name, invalid_id`

PROCEDURE

remove_shared_data

Description

The `remove_shared_data` procedure is used to remove a specified shared data instance from the shared database.

Syntax

```
void remove_shared_data(
 /* transaction : in transaction_type*/
 /* element_name : in string */
 /* id : in id_type */
);
```

Parameters

`transaction`	Transaction from which to remove the shared data element.
`element_name`	Name of element to be removed.
`id`	Existing shared data ID.

Status

`ok, out_of_memory, null_element_name, invalid_id`

PROCEDURE

rollback_transaction

Description	The `rollback_transaction` procedure is used to abort a given transaction and to delete the associated transaction buffer.

Syntax	`void rollback_transaction(`
	`  /* transaction : in transaction_type */`
	`);`

Parameters	`transaction` Existing transaction ID.
Returns	A handle to a newly-created element
Status	`ok, invalid_transaction_handle`

PROCEDURE

serpent_init

Description	The `serpent_init` procedure performs necessary initialization of the interface layer.

Syntax

```
void serpent_init(
 /* mailbox : in string */
 /* ill_file : in string */
);
```

Parameters		
	`mailbox`	`MAIL_BOX` constant defined in Saddle-generated include file.
	`ill_file`	`ILL_FILE` constant defined in Saddle-generated include file.

Status

```
ok, out_of_memory, null_mailbox_name,
null_ill_file_name, system_operation_failed
```

PROCEDURE

serpent_cleanup

Description	The `serpent_cleanup` procedure performs necessary cleanup of the interface layer.
Syntax	`void serpent_cleanup();`
Parameters	None.
Status	`ok`

PROCEDURE

set_undefined

Description

The `set_undefined` procedure sets the value of the data pointed to by value to undefined. The `set_undefined` procedure cannot be used with an entire shared data record at once.

Syntax

```
void set_undefined(
 /* type : in serpent_data_type */
 /* data : in caddr_t */
);
```

Parameters

`type`	The type of the shared data component.
`data`	Pointer to the value being set to undefined.

Status

`ok, operation_undefined_type`

PROCEDURE

start_recording

Description	The `start_recording` procedure enables recording. Once `start_recording` has been called, all transactions and associated data will be saved out to the specified file until the `stop_recording` procedure is invoked.

Syntax

```
void start_recording(
 /* file_name : in string */
 /* message : in string */
);
```

Parameters		
	`file_name`	File to which to write recording.
	`message`	Recording description.

Status `ok, io_failure, already_recording`

FUNCTION

start_transaction

Description	The `start_transaction` function is used to uefine the start of a series of shared data modifications.
Syntax	`transaction_type start_transaction();`
Parameters	None.
Returns	A unique transaction ID.
Status	`ok, out_of_memory, overflow`

PROCEDURE

stop_recording

Description	The `stop_recording` procedure causes the current recording to be stopped.
Syntax	`void stop_recording();`
Parameters	None.
Status	`ok, io_failure, invalid_process_record`

Appendix C Commands for Testing Serpent Applications and Dialogues

This appendix contains definitions of commands provided with Serpent to assist in testing Serpent applications and dialogues. The following is a list and short description of each of these commands. A more complete description immediately follows:

Command	Description
`format`	converts a recording file into an easy-to-read report
`playback`	used to play back a recording file

format

Description	The `format` command converts a binary Serpent transaction log to a formatted, easy-to-read report. The report is written to standards output.	
Definition	`format recfile`	
Parameters	`recfile`	The transaction log to be converted.
Returns	0	ok

COMMAND

playback

Description	The `playback` command can be used to reenact a session based on a recording file.	
Definition	`playback recfile host_mailbox correspondents`	
Parameters	`recfile`	The name of the file containing the recording to be played back.
	`host_mailbox`	The mailbox for the process to be simulated.
	`correspondents`	List of correspondents (the default is "all").
Returns	`0`	ok
	`1`	dialogue not found
	`2`	playback file not found
	`3`	error during playback

Appendix D Spider Example

```
/*-------------------------------------------------------------
----------*\
|
| Name: Spider Example
|
| Description:
|     Creates a spider chart for two well known correlation
centers and n
|     sensors defined in an external data file.
|
\*---------------------- Copyright 1987 CMU ----------------
----------*/

#define memoryPack

#include <signal.h>    /* UNIX signal handling */
#include <string.h>    /* C string functions              */
#include "serpent.h"   /* serpent interface definition    */
#include "hashPack.h"
#include "spiderA.h"   /* application data structures    */
#include "shared.h"    /* defs shared with dialogue       */

#define SDATA "sdata"
#define MAX_HASH 257

typedef struct {
 id_type self;
} generic_sdd;

boolean done = false;  /* main loop condition         */

/*-------------------------------------------------------------
----------*\
| Routine: sss_handle_interrupt
|
| Description:
|     Routine to handle the interrupt signal. Very UNIX
specific.
\*-------------------------------------------------------------
----------*/
void sss_handle_interrupt()
{
   fprintf(stderr, "spiderA.main interrupted. Exiting.\n");
   fflush(stderr);
   serpent_cleanup();
   check_status("sss_handle_interrupt:bad status from
serpent_cleanup.");
   exit(0);
}

/*-------------------------------------------------------------
----------*\
| Routine: sss_match_id
|
| Description:
|    Routine to compare an identifier with the id in a shared
```

```
data record.
\*-------------------------------------------------------------
----------*/
int sss_match_id(id, shared_data)
iid_id_type id;
generic_sdd *shared_data;
{                       /* local type definitions    */
 set_status(ok);        /* begin                       */

 return(shared_data->self == id);
}                               /* end sss_match_id    */

/*-------------------------------------------------------------
----------*\
| Routine: sss_hash_id
|
| Description:
|     Internal function which will convert an id used to index
|      in the array of hash lists.
\*-------------------------------------------------------------
---------*/
int sss_hash_id(id)
iid_id_type id;
{                       /* local type definitions    */
/*
** Initialize.
*/
 set_status(ok);        /* begin                       */
/*
** Return a value in the right range.
*/
 return((int)id % MAX_HASH);
}                               /* end sss_hash_id     */

/*-------------------------------------------------------------
----------*\
|  Test spiderA.main
|
|  Description:
|     Performs a write operation using ids.
\*-------------------------------------------------------------
----------*/
main()
{                       /* local variables             */
  transaction_type transaction;
  id_type id;

  communication_line_sdd *communication_line;
  sensor_sdd *sensor;

  int i;                /* random counter              */
  int sensor_count;     /* number of sensors           */

  HASH id_table;                /* hash of id's of all shared
data in view  */
  FILE *fp;             /* file pointer to the data file    */

  generic_sdd *shared_data_element;/* generic shared data
element ptr  */
/*
**  Initialize.
*/
```

```
  signal(SIGINT, sss_handle_interrupt);

/*
**  Initialize Serpent
*/
  serpent_init(MAIL_BOX, ILL_FILE);
  check_status("spiderA.main: bad status from
serpent_init.");
/*
**  Create an id hashtable.  This table contains the ids of
all the shared
**  data elements in the "view" at any given time.
*/
  id_table = make_hashtable(MAX_HASH, sss_hash_id,
sss_match_id);
  check_null(
    id_table,
    "dea.main:  out of memory.\n",
    out_of_memory
  );
/*
**  Open the data file.
*/
  fp = fopen(SDATA, "r");
  check_null(
    fp,
    "spiderA.main: could not open ill file.\n",
    system_operation_failed
  );
/*
**  Get the number of sensors from the data file.
*/
  fscanf(fp, "%d", &sensor_count);
/*
** Start a transaction.  Then, read in each sensor and
corresponding
** communication lines and put into shared data.
*/

  transaction = start_transaction();
  check_status("spiderA.main:  bad status from
start_transaction.");
  i = 0;
  while (i++ < sensor_count) {/* while sensor's left    */
/*
** Create shared data record.
*/
    sensor = (sensor_sdd *)make_node(sizeof(sensor_sdd));
    check_null(
      sensor,
      "spiderA.main: out of memory during make_node
sensor_sdd.\n",
      out_of_memory
    );
/*
**  Read in sensor data into shared data record.  Note: '&'
address
**  operators are omitted from certain structure elements
because they are
**  arrays and are already passed by reference.
*/
    fscanf(
      fp,
```

```
      "%s%d%s%s%s%s",
      sensor->site_abbr,
      &sensor->status,
      sensor->site,
      sensor->last_message,
      sensor->rfo,
      sensor->etro
    );
/*
** Put sensor into shared data and add to id table.
*/
    sensor->self = add_shared_data(
      transaction, "sensor_sdd", NULL, NULL
    );
    check_status("spiderA.main:  status from
add_shared_data.");

    put_shared_data(
      transaction, sensor->self,"sensor_sdd", NULL, sensor
    );
    check_status("spiderA.main:  status from
put_shared_data.");

    add_to_hashtable(id_table, sensor, sensor->self);
/*
**  Create communication line shared data element.
*/
    communication_line = (communication_line_sdd *)make_node(
      sizeof(communication_line_sdd)
    );
    check_null(
      communication_line,
      "spiderA.main: out of memory creating
communication_line_sdd.\n",
      out_of_memory
    );
/*
**  Read in communication line data into shared data element.
Note: '&'
**  operator is omitted from the 'etro' structure field since
it is an
**  array and is already passed by reference.
*/
    fscanf(
      fp, "%d%s", &communication_line->status,
communication_line->etro
    );

/*
**  Set from_sensor and to_cc fields.
*/
    communication_line->from_sensor = sensor->self;
    communication_line->to_cc = CMC_CODE;
/*
** Put communcation line into shared data and add to id table.
*/
    communication_line->self = add_shared_data(
      transaction, "communication_line_sdd", NULL, NULL
    );
    check_status("spiderA.main:  status from
add_shared_data.");

    put_shared_data(
```

```
      transaction,
      communication_line->self,
      "communication_line_sdd",
      NULL,
      communication_line
    );
    check_status("spiderA.main:  status from
put_shared_data.");

    add_to_hashtable(
      id_table, communication_line, communication_line->self
    );
/*
** Create communication line shared data element.
*/
    communication_line = (communication_line_sdd *)make_node(
      sizeof(communication_line_sdd)
    );
    check_null(
      communication_line,
      "spiderA.main: out of memory creating
communication_line_sdd.\n",
      out_of_memory
    );
/*
** Read in communication line data into shared data element.
Note: '&'
** operator is omitted from the 'etro' structure field since
it is an
** array and is already passed by reference.
*/
    fscanf(
      fp, "%d%s", &communication_line->status,
communication_line->etro
    );
/*
**  Set from_sensor and to_cc fields.
*/
    communication_line->from_sensor = sensor->self;
    communication_line->to_cc = OFT_CODE;
/*
** Put communcation line into shared data and add to id table.
*/
    communication_line->self = add_shared_data(
      transaction,
      "communication_line_sdd",
      NULL,
      communication_line
    );
    check_status("spiderA.main:  status from
add_shared_data.");

    add_to_hashtable(
      id_table, communication_line, communication_line->self
    );
  }                    /* end while file not empty
*/
/*
**  Close the data file.
*/
  fclose(fp);
/*
** Commit transaction.
```

```
*/
  commit_transaction(transaction);
  check_status("spiderA.main:  bad status from
commit_transaction.");
/*
** Go into wait loop for new transactions.
*/
  while (!done) {      /* do while not done           */

    transaction = get_transaction();
    check_status("spiderA.main:  bad status from
get_transaction.");

    id = get_first_changed_element(transaction);
    check_status("reader.main:  bad status from
get_first_changed_elem.");
/*
**  Process the elements in the transaction.
*/
    while (id != null_id) {/* while more changed elements
*/

     shared_data_element = (generic_sdd *)get_from_hashtable(
        id_table, id
      );
      check_null(
        shared_data_element,
        "spiderA.main: element not found in id_table
hashtable.\n",
        not_found
      );

      incorporate_changes(transaction, id,
shared_data_element);
      check_status("spiderA.main:  bad status from
incorporate_changes.");

      id = get_next_changed_element(transaction);
      check_status("spiderA.main: bad status from
get_next_chngd_elem.");

    }         /* end while more changed elements    */

  }           /* end while not done                 */
/*
**  Cleanup and return.
*/
  serpent_cleanup();
  check_status("spiderA.main: bad status from
serpent_cleanup.");

  return;
}
```

Index

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release Distribution Unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S CMU/SEI-91-UG-6	5. MONITORING ORGANIZATION REPORT NUMBER(S) CMU/SEI-91-UG-6
6a. NAME OF PERFORMING ORGANIZATION Software Engineering Institute 6b. OFFICE SYMBOL (if applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI Joint Program Office
6c. ADDRESS (City, State and ZIP Code) Carnegie Mellon University Pittsburgh PA 15213	7b. ADDRESS (City, State and ZIP Code) ESD/AVS Hanscom Air Force Base, MA 01731
8a. NAME OFFUNDING/SPONSORING ORGANIZATION SEI Joint Program Office 8b. OFFICE SYMBOL (if applicable) ESD/AVS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962890C0003
8c. ADDRESS (City, State and ZIP Code) Carnegie Mellon University Pittsburgh PA 15213	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO: 63752F; PROJECT NO.: N/A; TASK NO: N/A; WORK UNIT NO.: N/A
11. TITLE (Include Security Classification) Serpent: C Application Developer's Guide	
12. PERSONAL AUTHOR(S) User Interface Project	
13a. TYPE OF REPORT Final 13b. TIME COVERED FROM TO	14. DATE OF REPORT (Yr., Mo., Day) May 1991 15. PAGE COUNT ~90
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse of necessary and identify by block number)
FIELD	GROUP	SUB. GR.	Serpent, UIMS, user interface management system, user interface generators, C, application development

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Serpent is a user interface management system (UIMS) that supports the development and implementation of user interfaces, providing an editor to specify the user interface and a runtime system that enables communication between the application and the end user. This manual describes how to develop applications using Serpent. Readers are assumed to have read and underabstood the concepts described in the *Serpent Overview*, as well as to have had experience using the C programming language.

(please turn over)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ■ SAME AS RPT ☐ DTIC USERS ■	21. ABSTRACT SECURITY CLASSIFICATION Unclassified, Unlimited Distribution	
22a. NAME OF RESPONSIBLE INDIVIDUAL John S. Herman, Capt, USAF	22b. TELEPHONE NUMBER (Include Area Code) (412) 268-7630	22c. OFFICE SYMBOL ESD/AVS (SEI

ABSTRACT —continued from page one, block 19